VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# KNIHOVNA PRO INTERAKTIVNÍ NUMERICKÉ VÝPOČTY S RACIONÁLNÍMI ČÍSLY

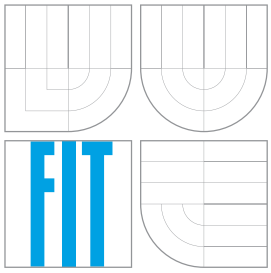
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL ŠPAČEK

BRNO 2014

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# KNIHOVNA PRO INTERAKTIVNÍ NUMERICKÉ VÝPOČTY S RACIONÁLNÍMI ČÍSLY

LIBRARY FOR INTERACTIVE NUMERICAL COMPUTATION WITH RATIONAL NUMBERS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL ŠPAČEK
AUTHOR

VEDOUCÍ PRÁCE doc. Ing. VLADIMÍR JANOUŠEK, Ph.D.
SUPERVISOR

BRNO 2014

# Abstrakt

Cílem této práce je návrh a implementace knihovny pro interaktivní numerické výpočty s racionálními čísly. Tato knihovna implementuje zeména numerické metody často používané v inženýrské praxi. Tato práce popisuje principy numerických metod vhodných pro toto použití společně s návrhem a implementací výsledné knihovny v jazyce Python.



# Abstract

The main aim of this thesis is to design and implement library for interactive numerical computation with rational numbers. The library implemenents mainly numerical methods commonly used in practical engineering. This thesis describes principles of numerical methods suited for computation using rational numbers and design and implementation of the library in Python language.



# Klíčová slova

Numerické výpočty, racionální čísla, maticové rozklady, řetězové zlomky, Python.

# Keywords

Numerical computation, Rational numbers, matrix decompositions, continued fractions, Python


# Citace

Michal Špaček: Knihovna pro interaktivní numerické výpočty s racionálními čísly, bakalářská práce, Brno, FIT VUT v Brně, 2014

# Knihovna pro interaktivní numerické výpočty s racionálními čísly

## Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Vladimíra Janouška, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Špaček
21. května 2014

## Poděkování

Chtěl bych poděkovat panu doc. Ing. Vladimíru Janouškovi, Ph.D. za vedení při vypracovávání bakalářské práce.

# Obsah

# Kapitola 1

# Úvod

Lidstvo využívalo výpočetní stroje pro ulehčení matematických výpočtů od nepaměti, ať již ve formě mechanických počítadel nebo později specializovaných výpočetních strojů (kalkulaček) a počítačů. Pro tyto výpočetní stroje je typické použití pevné přesnosti výpočtu, například kalkulačka s přesností na 10 míst by jako výsledek podílu $\frac{1}{3}$ vypočítala hodnotu 0.3333333333, což není přesný výsledek ale jen aproximace (přiblížení-se) přesnému výsledku, který nelze v omezené přesnosti vůbec vyjádřit. Použití pevné přesnosti je pro většinu aplikací opodstatněné kvůli menším výrobním nákladům (například u kalkulaček), jednodušší implementaci operací v pevné přesnosti a také faktu, že přesné výsledky nejsou velice často ani potřeba a chyba v řádu například $10^{-10}$ je zanedbatelná.

V některých situacích je ale potřeba přesných výsledků opodstatněná a chyby způsobené zaokrouhlování sem mohou velice snadno kumulovat a tím značně ovlivnit výsledek výpočtu. Proto je v této práci popsán návrh a implementace numerické knihovny používající aritmetiku s racionálními čísly.

V první kapitole Reprezentace desetinných čísel jsou popsány příčiny vzniku chyb zejména při použití výpočtů s pohyblivou řádovou čárkou a teorie řetězových zlomků a jejich použití pro aproximaci iracionálních konstant.

Druhá kapitola popisuje numerické výpočty v oboru racionálních čísel. Zejména teorii řetězových zlomků.

Třetí kapitola s názvem Maticové operace popisuje některé z maticových rozkladů, zejména těch vhodných pro použití s racionálními čísly, protože množství numerických metod využívá maticové rozklady (například řešení soustav lineárních rovnic nebo hledání inverze matice).

Čtvrtou kapitolou je Návrh řešení. Ta popisuje obecně strukturu a součásti, které by měla výsledná numerická knihovna mít, také jsou v ní popsány použité nástroje a současná řešení daného problému.

Pátá kapitola popisuje implementaci knihovny na základě návrhu z předchozí kapitoly.

Poslední kapitolou je závěr, ten shrnuje přínos práce a dosažené výsledky.

# Kapitola 2

# Reprezentace desetinných čísel

Pro reprezentaci desetinných čísel v počítači se v dnešní době nejčastěji používá norma IEEE 754. Reprezentovaná čísla jsou převáděna do normalizovaného tvaru[1], kde $s$ je znaménko reprezentovaného čísla, $m$ je mantisa, $b$ je základ zobrazovaného čísla (nejčastěji 2 nebo 10) a $e$ je exponent.

$$x = (-1)^s \times m \times b^e \tag{2.1}$$

Toto je základní normalizovaný tvar čísel podle IEEE 754. Z něj jsou odvozeny formáty pro uložení, lišící se v základu a rozsahu jednotlivých částí (pro přesnou reprezentaci čísel je nejdůležitější velikost mantisy). Ty jsou podrobněji popsány v[1]. Jde tedy o reprezentaci čísel používající pevnou přesnost, to znamená, že přesnost reprezentace čísel je omezena na fixní počet číslic. Opakem tohoto přístupu je reprezentace používající libovolnou přesnost, kdy je počet číslic pro uložení čísla omezen jen pamětí dostupnou na použitém výpočetním systému.

## 2.1 Chyby

Pro numerickou analýzu je typické získání přibližných výsledků (aproximací výsledku). Tyto přibližné výsledky jsou tedy oproti přesnému řešení zatíženy chybami, které mohou vznikat různými způsoby a také mohou být různými způsoby ovlivňovány. Některé z druhů chyb nelze na programové úrovni ovlivnit téměř vůbec, jiné lze ovlivnit jen velmi těžko, ale chyby určitého charakteru lze použitím jiných způsobů výpočtu téměř minimalizovat, čímž se výrazně zvětší přesnost výpočtu.

Pro každé zařízení realizující numerické výpočty existuje konečná množina čísel (označená například jako $A$), která je možné na daném zařízení reprezentovat. Pak ale nastává problém s reprezentací čísel takových, pro která platí $x \ni A$ pomocí jiného čísla $x' \in A$. Pro aproximované číslo $x' \in A$ pak musí platit[2]

$$|x - x'| \leqslant |x - g|, \ \forall g \in A \tag{2.2}$$

což znamená, že neexistuje jiné číslo bližší původnímu číslu $x$, které by jej aproximovalo lépe. Chyba vyjadřující přímo rozdíl mezi původním čísel $x$ a jeho aproximací $x'$ se nazývá absolutní chyba $E(x)$.

$$E(x) = |x - x'| \tag{2.3}$$

ta ale má nízkou vypovídající hodnotu například pro porovnávání chyb dvou řádově velice rozdílných čísel. Proto se častěji používá tzv. relativní chyba, vyjadřující podíl absolutní chyby k velikosti původního čísla.

$$E_{relative}(x) = \frac{|x - x'|}{x} \tag{2.4}$$

## 2.2 Druhy chyb

Chyby mohou na programové úrovni vznikat v různých fázích výpočtu vznikat více způsoby, což ovlivňuje rychlost jejich šíření, případně celkový vliv na konečný výsledek a tím i stabilitu výpočtu.

### 2.2.1 Chyby numerické metody

Chyby numerické metody ( *truncation errors* neboli "chyby odseknutí") vznikají nejčastěji jako důsledek převodu matematické úlohy na úlohu numerickou. Matematická úloha je definovaná jako výpočetní proces s nekonečným počtem kroků, převod na numerickou úlohu ale zahrnuje omezení počtu kroků na konečnou hodnotu, tak aby poskytoval aproximaci výsledku. K těmto chybám typicky dochází například při aproximaci funkcí pomocí nekonečných mocninných řad jako důsledek využití konečného počtu prvků řady, čímž je získaná aproximace výsledku, jejíž přesnost odpovídá počtu výpočetních kroků.

### 2.2.2 Zaokrouhlovací chyby

Jde o chyby způsobené převodem čísla na jeho reprezentaci v plovoucí řádové čárce. Převod čísla do tohoto tvaru může být proveden dvěma způsoby, buď zaokrouhlením, nebo odseknutím (podobně jako u chyb numerické metody). Zaokrouhlení se provádí následovně, mějme číslo $a$ reprezentované v desítkové soustavě.

$$|a| = 0, a_1\, a_2\, ...\, a_i\, a_{i+1}\, ...,\ 0\ \leqslant a_i \leqslant t, a_1 \neq 0 \tag{2.5}$$

Potom lze zaokrouhlené číslo $a'$ získat jako[6]

$$a' = \begin{cases} 0, a_1\, a_2\, ...\, a_t & if\ 0 \leqslant a_{t+1} \leqslant 4 \\ 0, a_1\, a_2\, ...\, a_t + 10^{-t} & if\ a_{t+1} \geqslant 5 \end{cases} \tag{2.6}$$

Výsledkem je sice zaokrouhlené číslo, to ale není v normalizovaném tvaru. Pro získání finální reprezentace čísla je potřeba provést normalizaci podle 2.1

$$x = sign(a) \times a' \times 10^b \tag{2.7}$$

Toto číslo už lze uložit na požadovanou přesnost, ale při výpočtu $a'$ dochází k mazání všech číslic po t. číslici. Původní hodnota smazaných číslic se může projevit jen na hodnotě poslední číslice (v případě zaokrouhlování) nebo vůbec (v případě odsekávání), kvůli čemuž vzniká zaokrouhlovací chyba. S zaokrouhlování/odsekáváním navíc souvisí pojem přesnost

stroje ( *machine precision* ). Přesnost stroje (označená jako $\epsilon$) používajícího aritmetiku v plovoucí řádové čárce lze vyjádřit pomocí vztahu[8]

$$\epsilon = b^{(1-p)} \qquad \text{Pro odsekávání} \tag{2.8}$$

$$\epsilon = \tfrac{1}{2} \times b^{(1-p)} \qquad \text{Pro zaokrouhlování} \tag{2.9}$$

kde $b$ je základ, v kterém je číslo uloženo (typicky 2) a $p$ je přesnost neboli počet bitů mantisy. Přesnost stroje ($\epsilon$) shora ohraničuje relativní chybu výpočtu a je možné ji také chápat jako nejmenší číslo zobrazitelné při dané přesnosti. Funkce poskytující reprezentaci v plovoucí řádové čárce je označena jako $fp(x)$.

$$\frac{|fp(x) - x|}{|x|} \leq \epsilon \tag{2.10}$$

### 2.2.3 Chyby přetečení/podtečení

K přetečení/podtečení dochází u čísel s příliš velkým exponentem (kladně nebo záporně), pokud by například na cílové architektuře bylo pro uložení exponentu (podle [1]) vyhrazeno v paměti $n$ bitů, ale pro uložení exponentu by byla potřeba větší paměť než byla vyhrazena, dojde k přetečení exponentu. V takovýchto případech je obvykle místo požadované hodnoty do paměti uložena speciální hodnota reprezentující chybu, ke které došlo a často dochází k zastavení výpočtu, protože se speciálními hodnotami nelze získat požadované výsledky a vznik této situace indikuje chybu ve výpočtu.

# Kapitola 3

# Šíření chyb při výpočtu

Výpočet pomocí určité numerické metody je prováděn pomocí konečného počtu základních aritmetických operací. Uspořádaná posloupnost základních aritmetických operací, která popisuje způsob jak pro daný vstup $x$ získat požadovaný výstup se nazývá algoritmus[2]. Pro algoritmus je definována množina vstupních hodnot $x$ a množina výstupních hodnot $y$. Pro každý krok výpočtu (označen jako $i$) se také dá definovat množina používaných operandů $o_i$, ta obsahuje buď hodnoty vstupní (ještě nepoužité) nebo hodnoty získané výpočty v některých z předchozích kroků algoritmů. Hodnoty z množiny $o_i$ mohou být pro kterýkoliv krok výpočtu zatíženy chybami. Nemusí ale jít jen o chyby, které vznikly při posledním kroku výpočtu, ale o chyby z některého z předchozích kroků výpočtu, proto je u algoritmu velice důležité pořadí operací a pro stejný výpočetní problém můžeme získat v závislosti na algoritmu (dekompozici a pořadí operací) výrazně odlišné výsledky. Toto kritérium pro hodnocení algoritmů se nazývá numerická stabilita algoritmu. Algoritmus je považován za numericky stabilní, pokud malá změna vstupních dat produkuje podobně malé změny v konečném výsledku. Jedno z nejdůležitějších kritérií pro hodnocení numerické stability je rychlost růstu chyby. Uvažujme počáteční chybu $E_0$ a chybu $E_n$ po $n$ krocích výpočtu. Pro numericky stabilní algoritmy bývá růst chyby typicky lineární[6], popsaný následujícím vztahem

$$E_n \approx C \times n \times E_0 \tag{3.1}$$

kde konstanta $C$ je nezávislá na počtu kroků. U numericky nestabilních algoritmů ale bývá růst chyby typicky exponenciální[6]

$$E_n \approx C^n \times E_0 \tag{3.2}$$

Z tohoto důvodu je nutné snažit se zmenšit rychlost růstu chyby. Toho může být dosaženo různými způsoby.

## 3.1 Změna algoritmu

Jeden z nejčastějších způsobů pro zmenšení chyby výpočtu je použití jiného algoritmu popisujícího výpočet. Odlišný algoritmus může například používat jiné pořadí výpočtu, díky čemuž dojde k menší kumulaci nebo i vyrušení chyb a tím bude finální výsledek přesnější. Další možností je použít algoritmus realizující jinou (stabilnější) numerickou metodu (často

na úkor jiných parametrů kvality výpočtu, například intervalu vstupních hodnot nebo složitosti algoritmu). Také může být použito nahrazení některých kroků výpočtů nebo operací jinými (ekvivalentními), které se oproti původním méně podílí na růstu chyby.

## 3.2 Minimalizace chyb vstupních dat

Vstupní data určují počáteční chybu $E_0$, pokud budou už vstupní data výrazně nepřesná, může být výsledek výpočtu velmi nepřesný, i když rychlost růstu chyby nebyla vysoká. V tomto případě sice nejde o minimalizaci rychlosti růstu chyby při výpočtu, ale chyby vstupních dat se mohou výrazně podílet na chybě výsledku.

## 3.3 Změna vnitřní reprezentace

K akumulaci chyb dochází nejvíce kvůli chybám v mezivýsledcích, k vzniku a rychlému růstu chyb by docházelo i v případě, kdy by všechna vstupní data byla zatížena nulovou nebo minimální chybou například kvůli dříve zmíněným zaokrouhlovacím chybám. Zmenšit rychlost růstu chyby a celkově zlepšit přesnost výpočtu tedy lze i změnou vnitřní reprezentace čísel (mezivýsledků). V případě nejčastěji používaného IEEE 754 je možné použít vyšší přesnost (formát double místo formátu single) případně rozšířenou dvojitou přesnost (formát *extended double*), u kterých je mimo jiné zdvojnásobená délka mantisy, což umožňuje přesnější ukládání reálných čísel i v případě mezivýsledků. Toto řešení s sebou samozřejmě nese problémy s větší paměťovou náročností výpočtu, což se může u výpočtů vyžadujících velké množství mezivýsledků (například operace s maticemi) zásadně projevit. Jednou z dalších možností je používat při výpočtech jinou reprezentaci, například ve formě racionálních čísel.

# Kapitola 4

# Numerické výpočty v oboru racionálních čísel

## 4.1 Reprezentace racionálních čísel

Pro numerické výpočty je možné místo dnes běžného ukládání ve tvaru používajícím pohyblivou řádovou čárku použít reprezentaci pomocí zlomku. Například reálné číslo 0.12345 lze pomocí zlomku vyjádřit jako $\frac{12345}{100000}$. Tímto způsobem je možné přesně vyjádřit jen všechna racionální čísla, pro iracionální čísla je nutné najít dostatečně přesnou aproximaci, případně se vyhnout algoritmům, u kterých se dá při výpočtu předpokládat nutnost použití iracionálních čísel (například odmocniny nebo goniometrické funkce).

Přímá reprezentace pomocí zlomku jako v příkladu výše je samozřejmě velice neefektivní, protože v případě potřeby uložit číslo s přesností $t$ by ve jmenovateli byla hodnota řádově $10^t$, což je pro výpočty i ukládání čísel nevhodné z důvodů paměťové (uložení velkých čísel) i časové (operace s velkými čísly) náročnosti. Navíc by vznikaly další chyby kvůli početním operacím s velkými čísly. Jeden ze způsobů reprezentace reálných čísel jsou řetězové zlomky.

### 4.1.1 Reprezentace pomocí řetězových zlomků

Nechť $x \in \mathbb{R}$, potom k němu existuje právě jeden řetězový zlomek ve tvaru $a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \dots}}$ případně ve zkráceném zápise $[a_0; a_1; a_2; ...; a_n]$, kde $a_0 \in \mathbb{Z}, a_1, a_2, ..., a_n \in N$. Je-li číslo $x$ racionální, má řetězový zlomek konečný počet prvků. Pro iracionální číslo by měl řetězový zlomek nekonečný počet prvků. Řetězové zlomky je tedy možné použít pro aproximaci iracionálních čísel pomocí čísla racionálního. Iracionální číslo je možná aproximovat různě přesně, v závislosti na počtu použitých prvků řetězového zlomku. Zlomky získané součtem konečného počtu prvků nekonečného řetězového prvku se nazývají sblížené zlomky. Při aproximaci iracionálních čísel tedy hledáme sblížený zlomek s co nejmenší odchylkou a zároveň nejmenším jmenovatelem. Například iracionální číslo $\sqrt{2}$ vyjádřené s přesností na 6 desetinných míst by odpovídalo zlomku $\frac{1414213}{1000000}$, přitom pomocí řetězových zlomků lze najít zlomek $\frac{1393}{985}$, který toto číslo aproximuje se stejnou přesností, ale čitatel i jmenovatel je řádově mnohem menší. Pro aproximaci iracionálního čísla ale nejsou vhodné všechny

sblížené zlomky řetězového zlomku, některé z nich zlepšují přesnost aproximace minimálně za cenu výrazného zvětšení jmenovatele. Jako příklad je číslo $\sqrt{2}$, za jeho referenční hodnotu je považovaná hodnota 1.41421356237. Následující tabulka popisuje přesnost a hodnoty jednotlivých sblížených zlomků.

| # | Zlomek | Hodnota | Rozdíl oproti přesné hodnotě |
|---|---|---|---|
| 1 | $\frac{1}{1}$ | 1 | 0.41421356237 |
| 2 | $\frac{3}{2}$ | 1.5 | 0.08578643763 |
| 3 | $\frac{7}{5}$ | 1.4 | 0.01421356237 |
| 4 | $\frac{17}{12}$ | $1.41\overline{6}$ | 0.0245310429 |
| 5 | $\frac{41}{29}$ | 1.413793103448 | 0.00042045892 |

Z tabulky je vidět, že například rozdíl mezi přesností $\frac{17}{12}$ a $\frac{41}{29}$ je značný, i když zlomky jsou přibližně stejně velké.

# Kapitola 5

# Maticové rozklady

Při numerických výpočtech s maticemi je často přímý výpočet za použití původní matice velice neefektivní, větší efektivity se dá dosáhnout použitím maticového rozkladu, jehož cílem je rozložit matici na součin jiných matic, díky tomu je možné problém dekomponovat a provádět výpočet efektivněji po menších částech nebo využít vlastností nově získaných matic (například horní/dolní trojúhelníková matice při výpočtu řešení soustav lineárních rovnic) pro urychlení, či zlepšení numerické stability výpočtu.

## 5.1 LU rozklad

LU rozklad hledá dvě matice $L$ (dolní trojúhelníková matice) a $H$ (horní trojúhelníková matice) pro které platí $A = LU$, kde $A$ je čtvercová matice s nenulovým determinantem. Pro výpočet lze použít dvě metody, Doolittlův rozklad, nebo Croutův rozklad. V případě Doolittleova rozkladu je výsledkem matice $L$ s jedničkami na hlavní diagonále. Při použití Croutovy metody jsou jedničky na hlavní diagonále matice $U$. Mezi těmito dvěma přístupy jsou určité rozdíly, ale jejich efektivita je porovnatelná[8]. Dále v textu je používán Doolittlův rozklad. Rovnice $A = LU$ lze rozepsat pro jednotlivé prvky matice.

$$A_{ij} = \sum_{p=1}^{min(i,j)} L_{ip}U_{pj}, \text{ kde } 1 \leq i, j \leq n \tag{5.1}$$

kde po úpravách získáme dvě rovnice pro výpočet matic $L$ a $U$.

$$L_{ij} = \frac{A_{ij} - \sum_{p=1}^{j-1} L_{ip}U_{pj}}{U_{ij}} \text{ kde } i > j \tag{5.2}$$

$$U_{ij} = A_{ij} - \sum_{p=1}^{i-1} L_{ip}U_{pj}, \text{ kde } \leq j \tag{5.3}$$

Z těchto dvou rovnic lze snadno odvodit algoritmus pro výpočet $LU$ rozkladu. Při výpočtu matice $L$ je ale třeba použít dělení prvky matice na hlavní diagonále, výpočet by tedy skončil neúspěchem, například pokud by se na hlavní diagonále objevila nula. Dalším problémem

může být vznik velkých zaokrouhlovacích chyb, pokud jsou prvky na hlavní diagonále blízko nuly. Řešením těchto problémů je prohození řádků/sloupců matice tak, aby byla matice buď diagonálně řádkově dominantní, pak platí.

$$|a_{ii}| > \sum_{j \neq i} |a_{ij}|, \text{ kde } i = 1..., n \tag{5.4}$$

Nebo diagonálně sloupcově dominantní.

$$|a_{jj}| > \sum_{j \neq i} |a_{ij}|, \text{ kde } j = 1..., n \tag{5.5}$$

To se provádí výpočtem permutační matice $P$, pro kterou platí $PA = LU$. Vynásobením původní matice $A$ permutační maticí $P$ tedy dojde k prohození řádků/sloupců původní matice $A$ tak, aby prvky na hlavní diagonále nezpůsobovaly numerickou nestabilitu. Pro výpočet permutační matice existují dva základní způsoby.

### 5.1.1 Částečné prohazování (*Partial Pivoting*)

Permutační matice je inicializována jako jednotková matice. Poté je v matici $A$ pro každý sloupec nalezen největší prvek (dle rovnosti 5.4). Na základě toho se v permutační matici $P$ provede prohození odpovídajícího řádku obsahujícího největší prvek s řádkem na hlavní diagonále. Časová náročnost této operace je $\mathrm{O}(n^2)$ [4].

### 5.1.2 Úplné prohazování (*Complete Pivoting*)

Permutační matice je stejně jako u částečného prohazování inicializována jako jednotková matice, ale pivot se nehledá jen v daném sloupci, ale v celé podmatici (podle rovností 5.4 a 5.1). Výsledkem jsou pak dvě permutační matice $P$ (řádkové prohazování) a $Q$ (sloupcové prohazování). Mezi permutačními maticemi $P$ a $Q$ a původní maticí $A$ pak platí $LU = PAQ$. Časová náročnost úplného prohazování je $\mathrm{O}(n^3)$ [6] proto se úplné prohazování nepoužívá velmi často kromě případů, kdy částečné prohazování nestačí k zajištění numerické stability.

Výpočet LU rozkladu tedy nevyžaduje využití odmocnin nebo jiných matematických operacích, které by mohly využívat iracionální čísla, proto je vhodný pro použití s racionálními čísly.

## 5.2 Využití LU rozkladu

### 5.2.1 Řešení soustav lineárních rovnic

Mějme soustavu lineárních rovnic v maticovém tvaru

$$Ax = b \tag{5.6}$$

kde $A$ je matice koeficientů, $x$ je vektor neznámých a $b$ je vektor pravých stran. Při použití LU rozkladu s částečným prohazováním lze tuto rovnici přepsat jako

$$LUx = Pb \tag{5.7}$$

Tato soustava se dá poté vyřešit posloupností dvou kroků[9].

1. Řešení spodní trojúhelníkové matice $Ly = Pb$, pro $y$ dopřednou substitucí
2. Řešení horní trojúhelníkové matice $Ux = y$ pro $x$ zpětnou substitucí

Oba tyto kroky se dají provést pro stejnou matici koeficientů přímo bez opakování výpočtu LU rozkladu a stejnou soustavu lze efektivně opakovaně řešit pro různé vektory $b$.

### 5.2.2 Inverzní matice

Mějme matici $A$ a matici k ní inverzní označenou jako $A^{-1}$. Pak musí platit

$$A \cdot A^{-1} = I \tag{5.8}$$

kde I je jednotková matice. Pro získání jednotlivých sloupců inverzní matice stačí vyřešit pro každý z nich soustavu $Ax = b$, kde $b$ jsou jednotlivé sloupce jednotkové matice[8]. Pro řešení těchto soustav lze využít LU rozklad za předpokladu, že je matice $A$ regulární.

### 5.2.3 Determinant

Determinant matice $A$ lze spočítat jako součin determinantů matic LU rozkladu takže $det(L) * det(U)$. Matice $L$ a $U$ jsou ale obě trojúhelníkové a pro trojúhelníkové matice lze jejich determinant spočítat jako součin prvků na hlavní diagonále[7] a protože všechny prvky matice $L$ na hlavní diagonále mají hodnotu 1 lze determinant spočítat pomocí vztahu

$$det(A) = \prod_{i=1}^{n} u_{ii} \tag{5.9}$$

V případě použití prohazování je ale potřeba spočítat také determinant permutační matice $P$, ten určuje jenom znaménko součinu podle počtu celkových výměn řádků.

$$det(A) = \prod_{i=1}^{n} u_{ii}(-1)^{v} \tag{5.10}$$

## 5.3 QR rozklad

QR rozklad je rozklad matice na $A$ na součin ortogonální matice $Q$ a horní trojúhelníkové matice $R$. Matice $A$ musí být čtvercová.

### 5.3.1 Grammův-Schmidtův algoritmus

Jeden ze způsobů výpočtu QR rozkladu matice je použití Grammova-Schmidtova ortogonalizačního procesu pro získání ortogonální matice $Q$. Tento proces je aplikován na vektory sloupců matice $A$ pro získání ortogonálních vektorů (sloupců matice $Q$). Mějme tedy bázi $a_1, a_2, ..., a_n$, kde jednotlivé vektory jsou sloupce matice $A$, výsledkem Grammova-Schmidtova algoritmu aplikovaného na tuto bázi je množina ortogonálních vektorů $u_1, u_2, ..., u_n$. Každý z těchto vektorů lze spočítat pomocí vztahu[4]

$$u_k = a_k - \sum_{i=1}^{k-1} proj_{e_i}(a_k); e_k = \frac{u_k}{\|u_k\|} \tag{5.11}$$

kde $proj_{e_i}(a_i) = \frac{<a_i,e_i>}{<a_i,a_i>}u$. Výsledkem je ortogonální matice $Q = [e_1, e_2, e_n]$ a horní trojúhelníková matice $R$, kde jsou jednotlivé prvky definovány jako $R_{ik} =< e,a_k >$ pro $i \leq k$. Tento postup se ale k praktickým výpočtům typicky nepoužívá, protože není dostatečně numericky stabilní[4]. Výhodnější je použít proces založený na Householderových transformačních maticích.

### 5.3.2 Householderovy transformační matice

Při výpočtu QR rozkladu pomocí Householderovy metody jsou voleny ortogonální matice $Q$ tak aby byly jejich postupnou aplikací nulovány prvky pod hlavní diagonálou vždy jen v jednom sloupci matice. Všechny Householderovy matice jsou tvaru

$$Q_k = \begin{bmatrix} I & 0 \\ 0 & F \end{bmatrix} \tag{5.12}$$

kde po vynásobení maticí $F_k$ musí být v $k$. sloupci nuly pod hlavní diagonálou. Vzorec pro výpočet Householderovy matice

$$F = I - 2\frac{vv^T}{v^Tv} \tag{5.13}$$

Vektor $v$ v tomto vzorci lze spočítat jako

$$v = x - \|x\|e_1 \tag{5.14}$$

kde $x$ je sloupec matice $A$ a $e_1$ je vektor $(1, 0, ..., 0)$. Výsledkem je Householderova matice $F_k$, která po aplikaci na původní matici $A$ dává

$$F_kA = \begin{bmatrix} \|x\| & * & \cdots & * \\ 0 & & & \\ \vdots & & A' & \\ 0 & & & \end{bmatrix} \tag{5.15}$$

Tento proces je postupně opakován pro všechny sloupce matice. Na konci výpočtu je matice $A$ v horní trojúhelníkové formě. Představuje matici $R$ v $QR$ rozkladu $A = QR$. Pro výpočet pomocí Householderových matic je potřeba $\tilde{2}mn^2 - \frac{2}{3}n^3$ operací[4] přičemž $n$ těchto operací je výpočet odmocniny, který je oproti ostatním operacím výrazně výpočetně náročnější kvůli racionálním číslům (podrobněji v kapitole 7.4.1).

### 5.3.3 Givensovy transformační matice

## 5.4 Využití QR rozkladu

**Řešení soustav lineárních rovnic**

Princip výpočtu je podobný postupu u $LU$ rozkladu 5.2.1. Soustavu lineárních rovnic $Ax = b$ lze zapsat jako $QRx = b$ kde po vynásobení obou stran maticí $Q^T$ je výsledkem $Rx = Q^T b$. Tato soustava je trojúhelníková a lze ji řešit pomocí zpětné substituce.

### 5.4.1 Choleského rozklad

Choleského rozklad je rozklad symetrické pozitivně definitní matice na součin matice spodní trojúhelníkové $L$ a transponované matice $L^T$. Algoritmus pro výpočet matice $L$ je možné odvodit ze vztahů[7]. Pro $i$. řádek

$$L_{ik} = \frac{A_{ik} - \sum_{j“1}^{k-1} L_{ij} Lkj}{L_{ii}} \tag{5.16}$$

$$L_{ii} = \sqrt{A_{ii} - \sum_{j“1}^{i-1} L_{ij}^2} \tag{5.17}$$

Výpočet tohoto rozkladu vyžaduje přibližně polovinu operací oproti výpočtu Gaussovy eliminace[4], nemůže být ale použit na libovolnou čtvercovou nesingulární matici. Počet operací na výpočet Choleského rozkladu je $\approx \frac{1}{3}m^3$. Z těchto operací je $m$ odmocnin, kvůli tomu může vznikat nepřesnost následkem jejich aproximace. Jedním z problémů je zjištění, je-li matice, na kterou je rozklad aplikován symetrická a pozitivně definitní. Symetrii matice lze lehce ověřit maximálně $n^2$ kroky, ověřovat ale pozitivní definitnost matice samostatně je poměrně složité, proto je vhodnější využít faktu, že při rozkladu indefinitní matice dojde k výpočtu odmocniny ze záporného čísla případně nuly. Tato situace lze snadno rozpoznat a na jejím základě ukončit výpočet.

### 5.4.2 $LDL^T$ rozklad

Jednou z variant Choleského rozkladu, která nepoužívá odmocniny je $LDL^T$ rozklad, kde $L$ je spodní trojúhelníková matice a $D$ diagonální matice. Oproti Choleského rozkladu jej lze použít i na symetrické ale indefinitní matice. Rozklad potom bude ve tvaru[5].

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ L_{21} & 1 & 0 \\ L_{31} & L_{32} & 1 \end{bmatrix} \begin{bmatrix} D_{11} & 0 & 0 \\ 0 & D_{22} & 0 \\ 0 & 0 & D_{33} \end{bmatrix} \begin{bmatrix} 1 & L_{21} & L_{31} \\ 0 & 1 & L_{32} \\ 0 & 0 & 1 \end{bmatrix} \tag{5.18}$$

Upravené vztahy pro výpočet jednotlivých prvků matic tedy jsou

$$D_{ii} = A_{ii} - \sum_{j“1}^{i-1} L_{ij}^2 D_{jj} \tag{5.19}$$

$$L_{ki} = \frac{A_{ki} - \sum_{j“1}^{i-1} L_{ij} L_{kj} Djj}{D_{ii}} \tag{5.20}$$

Jak Choleského tak $LDL^T$ rozklad se dají použít k řešení soustav lineárních rovnic podobně jako u LU rozkladu.

### 5.4.3 Řádkově odstupňovaný tvar matice

Matice $A \in \mathbb{R}^{m \times n}$ je v řádkově odstupňovaném tvaru (dále RREF z anglického *Reduced Row Echelon Form* ) pokud platí následující podmínky[7]

1. Prvních $r$ řádků matice je nenulových a řádky $r + 1 \to m$ jsou nulové
2. V každém nenulovém řádku je prvním nenulovým prvkem jednička (v sloupci $k$)
3. Ostatní prvky v sloupci $k$ jsou nulové

Matici lze převést do RREF pomocí Gaussovy eliminace, níže je algoritmus převodu podle[7].

1. j = 0
2. Pro každý řádek $i$
3. Hledáme první sloupec na pozici $> j$ s alespoň jedním nenulovým prvkem
4. Při nalezení takového sloupce $j$ = číslo sloupce a $x$ =číslo řádku s prvním nenulovým prvkem
5. Prohodíme řádky $x$ a $i$, aby na pozici pivotu (vedoucího prvku řádku) nebyla nula (pokud tam je)
6. Vydělíme každý prvek řádku $i$ hodnotou $\frac{1}{A_{ij}}$, což je hodnota na pozici pivotu, tím se na pozici pivotu dostane 1
7. Pro každý řádek $k$. Odečteme od řádku na pozici $k$ řádek na pozici $i$ vynásobený prvkem $A_{kj}$. Tím získáme nuly na hodnotách prvků pod vedoucím prvkem
8. j = j + 1 a pokračování prvním bodem, pokud není $j \geq n$

Po provedení algoritmu je výsledkem matice v RREF Matice převedená do RREF lze použít

$$\begin{bmatrix} 1 & 0 & * & * & 0 & * \\ 0 & 1 & * & * & 0 & * \\ 0 & 0 & 0 & 0 & 1 & * \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Obrázek 5.1: Matice v Redukovaném Řádkovém Schodovitém Tvaru (RREF)

například k řešení soustav lineárních rovnic. Jednou z dalších aplikací je výpočet hodnostního rozkladu matice.

## 5.5 Hodnostní rozklad

Nechť $A^R$ je RREF matice $A \in \mathbb{R}^{m \times n}$, pak každý rozklad $A = BC$ je hodnostním rozkladem matice[7]. Pokud $B \in \mathbb{R}^{m \times r}$ je matice s lineárně nezávislými sloupci a $C \in \mathbb{R}^{r \times n}$ je matice s lineárně nezávislými řádky a $r$ je počet nenulových řádků v $A^R$ (viz. 5.4.3). Pro konstrukci matic $B, C$ je použita matice $A^R$. Matice $B$ obsahuje lineárně nezávislé sloupce matice $A$, to jsou ty sloupce u kterých je v odpovídajícím sloupci v matici $A^R$ vedoucí jednička některého z řádků (pivot). Matice $C$ pak obsahuje nenulové řádky matice $A^R$, jejichž počet je stejný jako počet lineárně nezávislých řádků v matici $A$.

# Kapitola 6

# Návrh řešení

Cílem práce je vytvořit knihovnu pro interaktivní numerické výpočty s racionálními čísly. V této části bude popsán návrh řešení problému a jeho dekompozice s ohledem na splnění zadání práce, dále také nástroje (zejména programovací jazyk, prostředí a nutné součásti) pro tvorbu a použití řešení.

## 6.1 Existující řešení

Existuje celá řada počítačových algebraických systémů pro numerické nebo symbolické výpočty. Většina z těchto systémů ale používá primárně aritmetiku v plovoucí řádové čárce, ač často podporují i aritmetiku s racionálními čísly. V této části budou dále popsána všechna řešení významná ve vztahu k této práci.

### 6.1.1 SciPy

SciPy je open-source výpočetní prostředí jazyka Python, které obsahuje množství nástrojů zaměřených na různé oblasti vědeckých výpočtů a řadu podpůrných nástrojů. Níže je přehled základních součástí prostředí SciPy.

**NumPy**
: Numpy je rozšíření jazyka Python zabývající se implementací obecných N-rozměrných polí a efektivní manipulaci s nimi. Také poskytuje nástroje pro numerické výpočty (zejména Fourierovy transformace, lineární algebra, práce s náhodnými čísly) a zpracování velkých objemů dat. NumPy využívá pro výpočty FP aritmetiku a nemá implicitní podporu pro algebraické výpočty.

**SymPy**
: SymPy je systém pro počítačové algebraické (symbolické) výpočty, oproti NumPy který využívá hlavně numerické výpočty. Poskytuje nástroje pro řešení matematických problémů pomocí algebraické manipulace.

## 6.2 Nástroj pro řešení

Pro zajištění interaktivity s uživatelem je vhodné použít jazyk s interaktivním vstupem, ten lze sice využívat i u jazyků bez jeho implicitní podpory, ale to za cenu složitějšího použití (nutnost používat knihovny nebo moduly pro interaktivní vstup). Jeden z jazyků s interaktivním vstupem je jazyk Python, ten přímo v základním balíku obsahuje interpret s interaktivním režimem, což výrazně zjednodušuje používání výsledné knihovny. Python je navíc velice rozšířený jazyk s rozsáhlou sbírkou knihoven a podrobnou dokumentací. Další výhodou je neomezená přesnost celočíselného typu *long*(platí pro Python verze 2) případně *int*(v Pythonu verze 3 byly oba celočíselné datové typy *int* a *long* sjednoceny pod typ *int*, který se chová jako typ *long* v Pythonu verze 2).

## 6.3 Reprezentace racionálního čísla

Numerické výpočty s racionálními čísly typicky vyžadují datový typ pro reprezentaci racionálního čísla. V jazyce Python je pro tento účel použita knihovna *Fractions* [1], která implementuje všechny základní operace s racionálními čísly a navíc nabízí další funkce jak například aproximaci reálného čísla v FP formátu racionálním číslem nebo výpočet největšího společného dělitele Euklidovým algoritmem. Další možností je nevyužít standardní knihovny jazyka python a vytvořit vlastní třídu pro reprezentaci racionálního čísla, tato možnost je vhodná z několika důvodů. Jedním z nich je snadné zajištění kompatibility s třídou pro reprezentaci matice (zejména operací mezi nimi). Dalším z důvodů je výrazně větší kontrola při provádění operací s racionálními čísly jako například výběr vhodného způsobu výpočtu podle velikosti čísla (viz 7.2) nebo průběžné omezování velikosti jmenovatele.

## 6.4 Reprezentace matice/vektoru

Mezi knihovnami jazyka Python není třída pro reprezentaci matice, proto se jako nejlepší možnost jeví implementace vlastní třídy používající racionální čísla. Tato třída by měla uchovávat informace o všech prvcích matice (kde jednotlivé prvky budou racionální čísla) ve vhodném formátu a doplňující informace o matici, zejména její rozměry. Dále je potřeba aby implementovala základní operace mezi maticemi a poskytovala metody pro manipulaci s maticí nebo zjištění jejich vlastností.

[1] Dostupná v jazyce python verze 2.6 a vyšší

```
>>> r = Rational( 1, 3 ) * Rational( 2, 6 )
>>> print( r )
1/9
>>> r += 4
37/9
>>> a = Matrix( [ [ 3, 0, 8 ], [ 3, 5, 7 ], [ 7, 6, 2 ] ] )
>>> a.isSymetric()
False
>>> a.isSquare()
True
>>> b = a / 3
>>> print( b )
[1, 0, 8/3]
[1, 5/3, 7/3]
[7/3, 2, 2/3]
```

Obrázek 6.1: Ukázka využívání interaktivního vstupu při používání knihovny pro numerické výpočty

## 6.5 Knihovna pro numerické výpočty

Výsledná knihovna poté vhodným způsobem využívá tyto třídy pro numerické výpočty s racionálními čísly. Výpočty lze provádět buď interaktivně, nebo ve formě skriptů spouštěných bez interakce s uživatelem.

# Kapitola 7

# Implementace

## 7.1 Racionální čísla

7.3 Nejsnadnější reprezentace racionálního čísla $x = \frac{a}{b}$ je pomocí dvou celých čísel, která tvoří prvočíselnou dvojici. Při implementaci základních operací s racionálními čísly je potřeba, aby po provedení operace výsledné racionální číslo tvořilo prvočíselnou dvojici. To se dá zajistit pomocí výpočtu největší společného dělitele (dále $GCD(a, b)$). Při použití $GCD$ lze podle základní operace s racionálními čísly $x = \frac{a}{b}$ a $y = \frac{c}{d}$ definovat takto

$$x \cdot y = \left( \frac{(ac)}{GCD(ac, bd)}, \frac{(bd)}{GCD(ac, bd)} \right) \tag{7.1}$$

$$x/y = \left( \frac{(ad)}{GCD(ad, bc)}, \frac{(bc)}{GCD(ad, bc)} \right) \tag{7.2}$$

Analogicky lze definovat i součet/rozdíl.

$$x \pm y = \left( \frac{ad \pm bc}{GCD(ad \pm bc, bd)}, \frac{bd}{GCD(ad \pm bc, bd)} \right) \tag{7.3}$$

V této situaci ale může docházet k práci se zbytečně velkými čísly. Proto je vhodné oddělit situaci kdy $gd_1 = GCD(ad \pm bc, bd)$ je rovno jedné. K této situaci dochází podle[3] u náhodně volených čísel v přibližně 61 procentech případů. V těchto případech lze výsledek spočítat jako $(ad \pm bc, bd)$ podle. Pokud je ale $gd_1 > 1$ pak lze využít postupu

$$t = a(d/gd_1) \pm c(b/gd_1) \tag{7.4}$$

$$gd_2 = GCD(t, gd_1) \tag{7.5}$$

$$x \pm y = (\frac{t}{gd_2}, \frac{b}{gd_1}\frac{d}{gd_2}) \tag{7.6}$$

Takto je možné implementovat základní operace s datovým typem racionálního čísla.

## 7.2 Určení největšího společného dělitele

Pro implementaci základních operací s racionálními čísly je potřeba aby numerická knihovna implementovala funkce pro výpočet největšího společného dělitele (dále GCD z anglického *Greatest Common Divisor*). $GCD(a, b)$, kde $a$ a $b$ jsou nezáporná celá čísla vrací největší číslo, kterým je možné tato čísla vydělit beze zbytku. V této části bude představeno několik algoritmů realizujících jejich výpočet, popsány jejich základní vlastnosti a výsledky jednoduchého porovnání jejich efektivity při výpočtech s racionálními čísly.

### 7.2.1 Euklidův algoritmus

Euklidův algoritmus pracuje vždy s dvěma čísly. Na začátku jsou to vstupní čísla $a$, $b$, poté je v každém kroku vypočítán rozdíl mezi těmito dvěma čísly, ten je použit společně s menším z čísel v další iteraci jako výchozí čísla. Tento proces se opakuje do té doby, než jsou obě čísla stejná. Výsledné číslo je potom největší společný jmenovatel. Pro získání rozdílu mezi čísly je v moderní verzi algoritmu použito dělení modulo, efektivita algoritmu se tedy dá vyjádřit pomocí počtu kroků (podílů). Ten bude největší, budou-li vstupní čísla dva po sobě jdoucí prvky Fibonacciho posloupnosti a průměrný počet kroků je ale přibližně[3]

$$\frac{12 \ln 2}{\pi^2} \ln n \tag{7.7}$$

### 7.2.2 Lehmerův GCD algoritmus

Při použití Euklidova algoritmu na velmi velkých vstupních hodnotách je třeba využít výpočtů s libovolnou přesností (oproti běžně používané pevné přesnosti). Nutnost provádět v každém kroku podíl za použití libovolné přesnosti může negativně ovlivnit výkon, protože operace nad čísly v libovolné přesnosti jsou náročnější než operace v pevné přesnosti [3]. Jeden z algoritmů vycházející z Euklidova algoritmu se snahou snížit počet operací s čísly v libovolné přesnosti je Lehmerův algoritmus. Základní myšlenkou tohoto algoritmu je využití jen určitého počtu $k$ vedoucích číslic původních čísel $a$, $b$, tak aby bylo možné použít pevnou přesnost. Pro každá dvě vstupní čísla $(a, b)$ jsou zvoleny další dvě dvojice $(a', b'), (a'', b'')$ tak aby platilo

$$\frac{a'}{b'} < \frac{a}{b} < \frac{a''}{b''} \tag{7.8}$$

Potom lze provádět jednotlivé kroky Euklidova algoritmu simultánně na dvojicích $(a', b')$ a $(a'', b'')$. Porovnáváním podílů těchto podílů, tak aby platilo 7.8 lze zjistit kdy je třeba zastavit výpočet v pevné přesnosti. Výsledky jednotlivých kroků Euklidova algoritmu na dvojicích $(a', b')$ a $(a'', b'')$ jsou průběžně ukládány a v okamžiku, kdy se podíly liší, jsou aplikovány na původní čísla $(a, b)$ za pomocí aritmetiky s libovolnou přesností, poté se celý proces opakuje až do nalezení finálního výsledku.

### 7.2.3 Binární GCD algoritmus

Binární GCD algoritmus je algoritmus pro nalezení $GCD(a, b)$, kde $a$, $b$ jsou nezáporná celá čísla stejně jako například Euklidův algoritmus. Oproti Euklidovu algoritmu ale nevyužívá k tomuto výpočtu operaci podílu, ta je nahrazena jednoduššími operacemi, konkrétně

```
rand_numbers = []
for i in range( 0, N ):
rand_numbers1.append( random.ranrange( LOWER_BOUND, UPPER_BOUND )
    )
rand_numbers2.append( random.randrange( LOWER_BOUND, UPPER_BOUND
   ) )

start = time.time()
for i in range( N ):
gcd( rand_numbers1[i], rand_numbers2[i] )
end = time.time()
```

Obrázek 7.1: Skript pro měření dob běhů jednotlivých algoritmů

odečítáním, testováním sudosti/lichosti a půlením (aritmetický posun doprava). Tento algoritmus pracuje v několika fázích

1. Opakované půlení čísel $a$, $b$ aby alespoň jedno z nich bylo liché. Počet půlení je uložen jako $k$
2. Ustanovení proměnné $t$. Pokud je $a$ sudé pak $t = -b$, jinak $t = a >> 1$
3. Opakované půlení proměnné $t$ dokud není její hodnota lichá
4. Je-li $t > 0$ pak $a = t$ jinak $b = -t$. Tím je větší z čísel $a$, $b$ nahrazeno $|t|$
5. Odečtení $t = a - b$. Pokud $t \neq 0$ návrat na krok 3, jinak je výpočet u konce a výsledkem je $a \cdot 2^k$

### 7.2.4 Testování

Testování jednotlivých algoritmů probíhalo způsobem hledání GCD pro určitý počet náhodně generovaných čísel $n$ v rozsahu $LOWER\_BOUND$ a $UPPER\_BOUND$. Čísla byla vygenerovaná předem, poté byla měřena doba běhu pro výpočet $GCD$ jednotlivými algoritmy pro všechna náhodně vygenerovaná čísla. Testování bylo prováděno skriptem interpretovaném Pythonem verze 2.7.6. na procesoru Intel®Core™i5-3470, operační systém Win7 64bit. Níže je skript popisující způsob měření dob běhů jednotlivých algoritmů.

Podle tohoto srovnání vychází Euklidův algoritmus nejlépe i pro práci s velkými čísly[1]. Euklidův algoritmus je použit i v knihovně *Fractions* 6.3.

## 7.3 Implementace aritmetických operací

Pro praktickou implementaci aritmetických operací popsaných v části 7.3 by bylo možné použít instanční metody, takže například součet racionálního čísla *r1* s racionálním číslem *r2* by bylo možné provést například voláním metody *add* takto *r1.add( r2 )* . Po

[1] okolo 600 číslic

| | Čas běhu[s] pro 1000 dvojic čísel v daném rozsahu | | | |
|---|---|---|---|---|
| Algoritmus | Rozsah $\langle 1, 2^{20} \rangle$ | Rozsah $\langle 1, 2^{256} \rangle$ | Rozsah $\langle 1, 2^{1024} \rangle$ | Rozsah $\langle 1, 2^{2048} \rangle$ |
| Euklidův | 0.00099992 | 0.02999997 | 0.20499992 | 0.63000011 |
| Lehmerův $k = 8$ | 0.00999999 | 0.44499993 | 2.49099993 | 6.52600002 |
| Lehmerův $k = 16$ | 0.00500011 | 0.34000015 | 1.61100006 | 3.93499994 |
| Lehmerův $k = 32$ | 0.00300002 | 0.24700021 | 1.161999940 | 2.63399982 |
| Binární | 0.00500114 | 0.21499991 | 1.23499989 | 3.54200000 |

Tabulka 7.1: Výsledky porovnání algoritmů pro výpočet GCD

provedení této metody přímo v objektu reprezentujícím číslo *r1* k změnám odpovídajícím provedení operace součtu. Tento způsob ale poskytuje menší kontrolu nad prováděním výpočtu a zesložiťuje implementaci některých numerických metod, u kterých je třeba aby se změny neprojevily přímo na původních datech. Z těchto důvodů je vhodnější využít k implementaci přetěžování aritmetických operátorů pomocí speciálních metod popsaných v [10] a implementovat operace tak, aby jejich výsledek nebyl uložen v některém z operandů. Při použití této implementace by pak provedení součtu nad výše popsanými operandy mělo tvar $r1 + r2$, po provedení této operace by ale čísla *r1* a *r2* zůstala nezměněna.

## 7.4 Aproximace pomocí řetězových zlomků

I při použití aritmetiky s racionálními čísly je v některých případech nutné použít iracionální konstanty nebo reálná čísla v aritmetice s plovoucí řádkovou čárkou, proto je nutné mít možnost získat jejich racionální aproximace. V této kapitole budou popsány metody získání těchto racionálních aproximací, nejčastěji za použití řetězových prvků.

### 7.4.1 Řetězový zlomek odmocniny celého čísla

Výpočet odmocniny racionálního čísla lze rozložit na výpočet odmocnin dvou celých čísel a následného podílu. Níže je pseudokód algoritmu pro vygenerování řetězového zlomku odmocniny libovolného kladného celého čísla $n$.

```
a0 = floor( sqrt( n ) ) # zaokrouhleni dolu
cf = [] # prazdny seznam
if ( a0 = sqrt( n ) )
return [ a0 ]
a, m d = a0, 0, 1

do
m = a * d - m
d = ( n - m * m ) / d
a = ( a0 + m ) / d
cf.push_back( a )
while ( d != 1 )

return cf
```

Výsledkem tohoto algoritmu je seznam přirozených čísel reprezentujících řetězový zlomek, z nějž je jednoduše možné získat výsledný zlomek (racionální aproximaci odmocniny čísla $n$). Řetězový zlomek reprezentující iracionální číslo je nekonečný, proto je třeba zastavit výpočet při dosažení požadované přesnosti. Pro řetězový zlomek $[a_0, a_1, ..., a_n]$ lze sblížený zlomek $\frac{P_i}{Q_i}$ spočítat pomocí rekurzivního vztahu pro jmenovatel $p_i$ a čitatel $q_i$

$$p_i = a_i p_{i-1} + p_{i-2} \tag{7.9}$$

$$q_i = a_i q_{i-1} + q_{i-2} \tag{7.10}$$

Tento postup je ale značně nevýhodný, výpočet by byl tímto omezen počtem prvků řetězového zlomku, takže až po vypočtení finální hodnoty sblíženého zlomku by bylo možné posoudit, jestli má být výpočet již zastaven nebo ne (například na základě dosažení požadované přesnosti nebo velikosti jmenovatele výsledné racionální aproximace). Proto je vhodnější použít iterativní výpočet založený na maticové formě řetězových zlomků.

$$\begin{bmatrix} p_k & p_{k-1} \\ q_k & q_{k-1} \end{bmatrix} = \begin{bmatrix} a_0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} a_1 & 1 \\ 1 & 0 \end{bmatrix} \cdots \begin{bmatrix} a_k & 1 \\ 1 & 0 \end{bmatrix} \tag{7.11}$$

Jednotlivé sblížené zlomky jsou získány násobením dílčích matic, tento způsob tedy umožňuje získávat mezivýsledky a podle nich výpočet zastavit výpočet při splnění podmínek.

### 7.4.2 Iracionální konstanty

Třída *Rational* také definuje racionální aproximace základních iracionálních konstant. Konkrétně $\pi, e, \sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{6}, \sqrt{7}, \sqrt{8}, \sqrt{10}$. Tyto aproximace jsou získány z řetězových zlomků pro každou z těchto konstant a jsou vybrány tak, aby zajišťovaly dostatečnou přesnost při rozumné velikosti jmenovatele. Pojmenování konstant v knihovně odpovídá jménům těchto konstant, tedy *PI, E, SQRTN* ( kde $N \in 2, 3, 5, 6, 7, 8, 10$ ). Pro eulerovo číslo a odmocniny jsou navíc implementovány funkce vracející seznam prvních $n$ členů řetězového zlomku. Ten lze pak převést na racionální číslo pro vyšší přesnost aproximace příslušné konstanty.

### 7.4.3 Racionální aproximace reálného čísla

Mějme reálné číslo $r$, kde $r_0$ značí první číslici tohoto čísla. Řetězový zlomek tohoto čísla jde v maticovém formátu vyjádřit podobně jako v sekci 7.11. Postup pro získání jednotlivých sblížených zlomků je následující. Nechť $m$ je matice obsahující součin prvních $k$ prvků řetězového zlomku, tedy sblížený zlomek $\frac{P_k}{Q_k}$. Pro $k = 0$ je $m = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$. Sblížený zlomek $\frac{P_{k+1}}{Q_{k+1}}$ lze pak získat jako součin

$$m = m \cdot \begin{bmatrix} r_0 & 0 \\ 1 & 0 \end{bmatrix} \tag{7.12}$$

Po tomto výpočtu je ještě třeba upravit číslo $r = \frac{1}{r - r_0}$ jako přípravu pro další krok výpočtu. Tento proces jde opakovat dokud $r \neq 0$, případně dokud není překročen limit na jmenovatele. Po ukončení iterace je zbylá hodnota $r$ v rozmezí $\langle 0, \frac{1}{r_0} \rangle$, to lze aproximovat buď jako 0, nebo $\frac{1}{maxdenominator}$. Algoritmus tedy vyzkouší obě možnosti a zvolí tu z nich, u které je menší odchylka proti číslu $r$.

## 7.5 Reprezentace matice

Pro reprezentaci matice jsem podle sekce 6.4 naprogramoval třídu *Matrix* , kde základ tvoří dvojrozměrné pole racionálních čísel a dvě celočíselné proměnné pro uložení rozměrů matice. Tato třída implementuje kromě numerických metod (maticových rozkladů) i základní operace a podpůrné metody pro manipulaci s maticemi. Většina popisovaných operací využívá obdobnou implementaci jako operace nad racionálními čísly (viz 7.3) kromě některých, u těch je ale objasněn důvod této odlišnosti. Podrobnější popis metod a jejich rozhraní poskytuje příloha B.

### 7.5.1 Základní aritmetické operace

Třída *Matrix* implementuje základní aritmetické operace mezi maticí a skalárním číslem a také operace mezi maticemi včetně maticového součinu.

### 7.5.2 Vlastnosti matice

Tato skupina metod zjišťuje na základě záznamů, případně rozměrů matice její vlastnosti, například symetrii, diagonalitu, schodovitost, atd. Všechny tyto metody neprovádějí žádné přímé změny v matici, jejich výsledkem je jen pravdivostní hodnota.

### 7.5.3 Řádkové/sloupcové operace

Řádkové a sloupcové operace jsou násobení řádku/sloupce konstantou a přičtení jednoho řádku k druhému. Výsledkem těchto metod jsou na rozdíl od jiných změny přímo v původní matici. Tyto metody jsou hojně používány například u LU rozkladu nebo převodu matice do RREF, je ale možné je používat i samostatně.

### 7.5.4 Manipulace s částmi matice

Manipulace zahrnuje zejména řádky/sloupce matice, jejich prohazování, mazání, nebo také výpočet normy určitého sloupce. Dále mezi tyto operace patří metody na získávání podmatic.

# Kapitola 8

# Závěr

V souladu se zadáním bakalářské práce jsem nastudoval běžně používané numerické metody a možnosti jejich použití pro numerické výpočty s racionálními čísly. Nastudované numerické metody byly posuzovány zejména podle vlivu použití racionálních čísel na složitost výpočtu a jejich numerickou stabilitu. Podle nastudovaných numerických metod a požadavků na výslednou knihovnu vyplývající ze zadání jsem provedl návrh výsledné numerické knihovny. Pro implementaci aritmetiky s racionálními čísly jsem nastudoval způsoby výpočtu největšího společného jmenovatele a jeho použití při implementaci této aritmetiky, také jsem provedl malé srovnání těchto algoritmů při práci s velkými čísly (jaká se často vyskytují při výpočty s racionálními čísly). Dalším krokem byl návrh a implementace způsobu reprezentace matic s racionálními prvky a numerických metod nad těmito maticemi (zejména maticových rozkladů). Z důvodů efektivní práce s racionálními čísly jsem implementoval metody pro aproximaci odmocnin celých čísel a reálných čísel racionálními čísly.

Pro některé z implementovaných numerických metod byly vytvořeny příklady jejich použití, které měly za účel demonstrovat použití knihovny v prostředí implementačního jazyka. Tyto vytvořené příklady tvoří jednu z příloh práce.

# Literatura


[1] Goldberg, D.: What Every Computer Scientist Should Know About Floating-Point Arithmetic. [online], 1991 [cit. 2014-05-10].

[2] Josef Stoer, R. B.: *Introduction to Numerical Analysis*. Springer, 2002.

[3] Knuth, D. E.: *The Art of Computer Programming, vol. 2 Seminumerical algorithms*. Addison-Wesley.

[4] Lloyd N. Trefethen, D. B.: *Numerical Linear Algebra*. SIAM: Society for Industrial and Applied Mathematics, 1997.

[5] Nguyen, D.: Cholesky and $LDL^T$ Decomposition. [online], 2010-07-29 [cit. 2014-05-16].
URL
http://mathforcollege.com/nm/mws/gen/04sle/mws_gen_sle_txt_cholesky.pdf

[6] Richard Burden, J. F.: *Numerical Analysis*. Cengage Learning, 2010.

[7] Rohn, J.: *Linární algebra a optimalizace*. Karolinum, 2004, ISBN 80-246-0932-0.

[8] Steven C. Chapra, R. P. C.: *Numerical methods for engineers*. McGraw-Hill, 1988.

[9] WWW stránky: Solving Fully Determined Systems. [online], 2012-03-01 [cit. 2014-05-15].
URL https:
//vismor.com/documents/network_analysis/matrix_algorithms/S3.SS1.php

[10] WWW stránky: Special method names. [online], 2014-05-18 [cit. 2014-05-20].
URL
https://docs.python.org/2/reference/datamodel.html#special-method-names

# Příloha A

# Instalace

Knihovnu lze používat v prostředí Python. Testována byla na pythonu verze 2.7. Pro používání knihovny je připraven skript `setup.py` . Jeho spuštěním s parametrem install se provede instalace skriptů, tak aby mohly být běžně používány v interaktivním prostředí.

```
python setup.py install
```

# Příloha B

# API

| | |
|---|---|
| `Rational( 3, 4 )` | Vytvoření racionálního čísla $\frac{3}{4}$. |
| `Rational( 3 )` | Vytvoření čísla 3 |
| `cf2rational( cont_fraction, denom )` | Převod řetězového zlomku na racionální číslo. `cont_fraction` je seznam prvků řetězového zlomku, `denom` je maximální velikost jmenovatele. |
| `real2rational( r, denom )` | Aproximace reálného čísla `r` racionálním číslem s jmenovatelem $<$ `denom` |
| `cf_sqrt2( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{2}$ |
| `cf_sqrt3( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{3}$ |
| `cf_sqrt5( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{5}$ |
| `cf_sqrt6( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{6}$ |
| `cf_sqrt7( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{7}$ |
| `cf_sqrt8( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{8}$ |
| `cf_sqrt10( n )` | Vrací seznam prvních `n` prvků řetězového zlomku pro $\sqrt{10}$ |
| `cf_euler( n )` | Vrací seznam prvních `n` prvků řetězového zlomku eulerova čísla |

Tabulka B.1: Tabulka funkcí pro obecnou práci s racionálními čísly

| | |
|---|---|
| `Matrix( [ [ 1, 2 ], [ 2, 1 ] ] )` | Vytvoření matice výčtem prvků. |
| `Matrix( 3, 3 )` | Vytvoření matice pomocí rozměrů. Prvky matice nastaveny na 0. |
| `m.identity()` | Naplní prvky matice tak, aby byla jednotková. Provádí změny v původní matici. |
| `m.isSymetric()` | Zjistí, je-li matice symetrická. Vrací booleovskou hodnotu. |
| `m.isDiagonal()` | Zjistí, má-li matice mimo hlavní diagonálu nenulové prvky. |
| `m.isTransposeOf( m2 )` | Zjistí, je-li matice `m` transpozicí vstupu(matice `m2` ). |
| `m.isSquare()` | Zjistí, je-li matice čtvercová. |
| `m.getTranspose()` | Vrátí transpozici matice `m` . Neprovádí změny v matici `m` . |
| `m.transpose()` | Provede transpozici matice m. Provádí změny v matici `m` . |
| `m.print_float()` | Zobrazí prvky matice jako podíly v plovoucí řádové čárce místo racionálních čísel. |
| `m.fill( val )` | Naplní matici hodnotou `val` . |
| `m.scale_row( r, x )` | Vynásobí řádek `r` konstantou `x` . Provádí změny v matici `m` |
| `m.scale_col( c, x )` | Vynásobí sloupec c konstantou x. Provádí změny v matici `m` |
| `m.add_row( a, b )` | Přičte řádek a k řádku b. Provádí změny v matici `m` |
| `m.add_col( a, b )` | Přičte sloupec a k sloupci b. Provádí změny v matici `m` |
| `m.swap_cols( x, y )` | Prohodí sloupce x a y. Provádí změny v matici `m` |

Tabulka B.2: Tabulka popisující základní funkce pro práci s maticemi

| | |
|---|---|
| `m.rref()` | Spočítá RREF matice. Výsledek vrací jako novou matici. |
| `m.rank_decomposition()` | Spočítá hodnostní rozklad matice. Výsledkem je dvojice matic v pořadí ( B, C ) |
| `m.lu()` | Vypočte LU rozklad matice `m` . Výsledkem je trojice matic v pořadí (L, U, P). Reference na matice rozkladu jsou uloženy i v objektu `m` jako instanční proměnné `L, U, P` |
| `m.solve_lu( right_side )` | Vyřeší soustavu rovnic, kde `m` je matice koeficientů a `right_side` je vektor pravé strany. Využívá instanční proměnné pro uložení výsledku LU rozkladu, případně zavolá metodu pokud jsou prázdné `m.lu()` |
| `m.inverse()` | Získá inverzní matici k matici `m` pomocí LU rozkladu. Stejně jako metoda `solve_lu()` využívá instanční proměnné pro uložení výsledku LU rozkladu. Vrací novou inverzní matici |
| `m.det()` | Vypočítá pomocí LU rozkladu determinant matice `m` . Vrací celočíselnou hodnotu. |
| `m.cholesky()` | Vypočítá Choleského rozklad matice. Výsledkem je symetrická matice `R` . |
| `m.solve_cholesky( right_side )` | Vyřeší soustavu rovnic. Stejné chování jako `m.solve_lu()` ale využívá Choleského rozklad. |
| `m.ldlt()` | Vypočte LDLT rozklad matice. Výsledkem je dvojice nových matic v pořadí ( L, D ). |
| `m.qr()` | QR rozklad matice |

Tabulka B.3: Tabulka popisující funkce pro maticové rozklady a numerické metody

# Příloha C

# Příklady práce s knihovnou

V této příloze budou ukázky práce s numerickou knihovnou ve formě skriptů, případně ukázek reakcí interaktivního vstupu. Pro všechny ukázky jsou příslušné skripty uložené ve složce `examples`

| | |
|---|---|
| Hodnostní rozklad | Odpovídající skript: `examples/rank_decomposition.py` |
| LU rozklad | Odpovídající skript: `examples/lu_decomposition.py` |
| Racionální čísla | Odpovídající skript: `examples/rational_sqrt.py` |

Tabulka C.1: Tabulka umístění skriptů k příkladům

## C.1 Hodnostní rozklad

Na tomto příkladu je na ukázkové matici $A$ proveden převod do RREF. Poté je získan hodnostní rozklad a násobením výsledných matic ověřena jeho správnost.

```
>>> A = Matrix( [ [ 1, 3, 1, 4 ], [ 2, 7, 3, 9 ],
                  [ 1, 5, 3, 1 ], [ 1, 2, 0, 8 ] ] )
>>> A.rref()
[1,      0,      -2,      0]
[0,      1,      1,       0]
[0,      0,      0,       1]
[0,      0,      0,       0]
>>> rank_decomp = A.rank_decomposition()
>>> rank_decomp[0] # Matice B
[1,      3,      4]
[2,      7,      9]
[1,      5,      1]
[1,      2,      8]
>>> rank_decomp[1] # Matice C
[1,      0,      -2,      0]
[0,      1,      1,       0]
[0,      0,      0,       1]
>>> rank_decomp[0] * rank_decomp[1] # A = BC
[1,      3,      1,       4]
[2,      7,      3,       9]
[1,      5,      3,       1]
[1,      2,      0,       8]
>>> ( rank_decomp[0] * rank_decomp[1] ) == A # Overeni rovnosti
True
```

Obrázek C.1: Ukázka práce s RREF rozkladem

## C.2 LU rozklad

V dalším příkladu je na vzorové matici proveden LU rozklad, ten je pak používán pro další výpočty.

```
>>> B = Matrix( [ [ 2, 5, 7 ], [ 6, 8, 3 ], [ 1, 2, 6 ] ] )
>>> B.lu()
(
[1,     0,     0]
[1/3,   1,     0]
[1/6,   2/7,   1]
,
[6,     8,     3]
[0,     7/3,   6]
[0,     0,     53/14]
,
[0,     1,     0]
[1,     0,     0]
[0,     0,     1]
)
>>> ( B.L * B.U ) == ( B.P * B ) # Overeni rovnosti
True
>>> B.solve_lu( [ 3, 6, 3 ] ) # Vyreseni pro danou pravou stranu

[93/53]
[-39/53]
[24/53]

>>>  B.inverse() # Vypocteni inverzni matice

[-42/53,        16/53,   41/53]
[33/53, -5/53,   -36/53]
[-4/53, -1/53,   14/53]
>>> B.det() # Vypocteni determinantu
-53
```

Obrázek C.2: Příklad výpočtů s LU rozkladem

## C.3 Odmocnina a práce s racionálními čísly

Následující skript generuje racionální aproximace odmocniny čísla 23 s postupně zvyšujícímse limitem pro velikost jmenovate. Poté porovnává chyby těchto aproximací oproti výsledku z standardní knihovny *math.sqrt*

```
from math import sqrt
from rational import *

sqrt_fp = math.sqrt( 23 )
for i in( 10, 100, 1000, 10000 ):
  sqrt_rational = sqrt( 23, i )
  print( sqrt_rational )
  print( "Denominator size limit: " + str( i ) )
  print( "Error compared to math.sqrt: "
  + str( sqrt_fp - float( sqrt_rational ) ) )
```

Obrázek C.3: Skript pro postupný výpočet racionálních aproximací odmocniny

```
>>>
24/5
Denominator size limit: 10
Error compared to math.sqrt: -0.00416847668728
235/49
Denominator size limit: 100
Error compared to math.sqrt: -8.68440342199e-05
1151/240
Denominator size limit: 1000
Error compared to math.sqrt: -1.81002061428e-06
43949/9164
Denominator size limit: 10000
Error compared to math.sqrt: 8.69028315975e-09
```

Obrázek C.4: Výstup skriptu